Upgrading Legacy Instances of Reactive Systems

Robert J. Hall
AT&T Labs Research
180 Park Ave, Bldg 103
Florham Park, NJ 07932
bob-2upgrading-@channels.research.att.com

Abstract

A software product typically goes through many "upgrades" (version changes) over its lifetime. Reactive systems, such as email clients, software agents, proxies, traffic controllers, and telephone switches are no exception. Evolving such stateful systems is made difficult by the fact that new versions of the software must deal correctly with legacy instances. *Users of earlier versions have invested significant resources in creating the state of the legacy instance, and usually require that this state be* upgraded *appropriately when the new system version is activated. However, validating the correctness of this upgrading behavior is particularly difficult, whether through testing or more formal techniques like model checking, because legacy states are typically unreachable to the new version of the software. This paper explores this problem and requirements for its solution; presents a simple conceptual and modeling/programming* upgrade framework*, based upon the idea of a* supermodel *that allows upgrade behavior to be validated using mainstream approaches; and gives techniques for simplifying the validation problem.*

1 Introduction

Software systems evolve through the installation of successive versions (*upgrades*) as errors are fixed and requirements change. When the software manages a persistent state, this introduces particular challenges, because the new version must correctly handle the fact that it is applied to a *legacy instance*, that is, an instance of a previous version of the system that resides in a particular state in the state space of the old version. Typically, on first activation after the upgrade, the new version will convert the state of the legacy instance to a semantically reasonable state in the new version's state space before proceeding with normal operations. This conversion may require supplying default values for new data fields, format conversions, data reorganization, or deletion of obsolete data elements. If this does *not* happen, or happens incorrectly, it can lead to significant loss of time and other resources as well as high user frustration. Here are a few examples.

- Incorrect upgrading of an electronic mail client can result in having to manually re-enter the names and addresses of all correspondents into the address book;
- Incorrect upgrading of a legacy "electronic wallet" could result in loss of the stored monetary value in the wallet;
- Incorrect upgrading of a file encryption utility could result in loss of the encrypted data due to incorrect conversion of existing files. If the error occurred only in infrequently visited files, it might not be discovered until too late to restore from backups.

Reactive systems comprise a broad class of stateful software systems: those which can be thought of as finite- or infinite-state machines that change state in response to *input events* and yield a set of *output events* on each transition. I will term the problem of correctly mapping a state from an older version's space into that of a newer version, the *state upgrade* problem.

In this paper, I present a conceptual and design framework that enables the validation of state upgrade behaviors, as well as some particular validation techniques for such behaviors.

1.1 Validating State Upgrades: The Problem

What makes validating state upgrade behaviors more difficult than validating "typical" system behaviors? That is, why can we not simply apply the same validation techniques to upgrade behaviors as we use for other behaviors?

First of all, legacy states are often *unreachable* by the new version. A legacy state is unreachable if starting from the default initial state of the new version, no sequence of input events will reach it. In the email address book example, suppose the new version adds the ability to keep track of an "encryption key" for each addressee. Then it is presumably impossible to reach a legacy state (starting from the default initial state of the new version) where a correspondent fails to have a defined value for this field. This is because the new version of the system, if correct, will always initialize it.

This unreachability hinders validation. Test coverage tools will report parts of the model implementing upgrade behaviors as uncovered; assertion checkers may not be exercised; and model checkers cannot discover errors in unreachable states, because they operate by searching only the reachable part of the state space for states satisfying a given condition. Since these states are unreachable in the code version of interest, older versions must be kept around and used to generate legacy states for input to the new version; for testing non-upgrade behaviors, by contrast, one need only maintain and test the latest version of the code.

And yet, this straight-forward approach of simply storing all versions and using them to generate legacy instances for testing is unwieldy. Not only does it require storage space linear in the number of versions, the size of an individual version grows linearly with the number of versions as well, because each version must have the code necessary for upgrading each prior version's states. This implies that the true storage requirement is quadratic in the number of versions. Clearly, any approach based on keeping around all versions for validation purposes (to form what I will define later as a "supermodel" of the system) must address this problem.

Another difficulty in validating upgrade behaviors is that unless carefully designed, the overall state space can increase exponentially in the number of system versions. The worst case of this is when each upgrade transition from any prior version leads to a distinct unreachable state in a higher version's state space. If this happens, then all previous state spaces are mirrored in a portion of each new version's state space that is disjoint from the part reachable by the new version. This is shown for four successive versions in Figure 1: if each version has a reachable space of n states (the doubly ringed subspaces), then version k's space could have $2^k n$ states, potentially requiring 2^k times as much effort to validate as it would if the system only had to operate on non-legacy instances. Of course, the blowup may be worse if successive versions have increasingly many reachable states. Note that, as is typical, we assume that every new version must be able to upgrade any older version's states, because not all users get all upgrades.

The consequence of this space blowup is that even if the system's non-upgrade behaviors can be shown to be totally correct on the reachable portion of the space, using standard

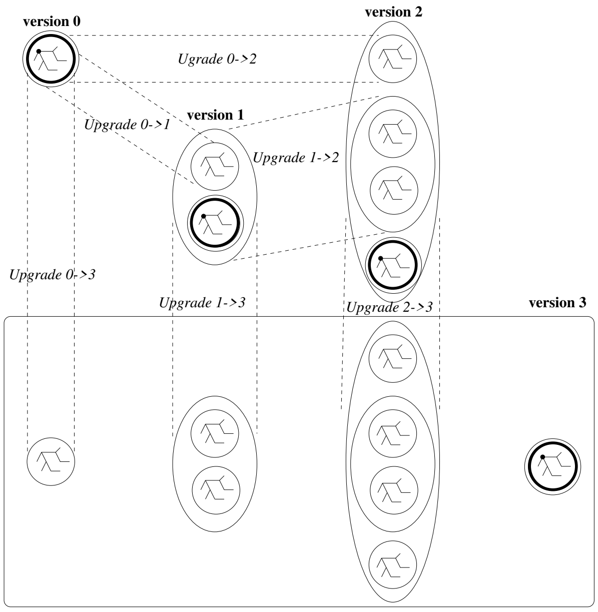


Figure 1. The state-space multiplication problem. Each circle represents a state (sub)space. The heavy dots represent the default initial state for the version. Doubly ringed (sub)spaces are the reachable portions. Dashed lines show worst-case upgrade-induced embeddings.

validation techniques, its correctness may depend upon invariants and other properties which are violated in states reachable only through upgrading. In fact, even if *every* version is totally correct on non-upgrade behaviors, an incorrect upgrade transition can result in subtle errors. This problem is exacerbated when earlier versions have errors that may have resulted in legacy instances that fail to obey expected invariants.

1.2 Contributions of This Paper

I first describe a set of goals and requirements for a conceptual and design framework addressing the upgrade problem. I then describe a novel *upgrade framework* meeting these requirements, which can be used to structure reactive system specifications, models, and implementations so as to enable and facilitate validation. I then discuss several validation techniques that can be applied within this framework to help with validating upgrade behaviors. Throughout I illustrate these ideas with examples drawn from a moderate sized, real-world (i.e., multiple versions deployed to users) case study involving an electronic mail agent system.

2 Requirements for an Upgrade Framework

Of course, systems are regularly deployed today with upgrade behaviors, and presumably they are validated to some degree prior to deployment. The goal of this work is to clarify and systematize this process to achieve

- *Wide applicability:* the framework and validation techniques should be applicable to a broad class of systems, independent of domain, level of description, and specification or programming language. It should scale to all sizes of software system.
- *Testability:* The framework should enable and facilitate software testing and testing support tools such as (a) specification or code coverage tools, (b) assertion checking[8], and (c) test generation tools.
- *Formal methods applicability:* The framework should support the application of formal methods tools such as model checkers and theorem provers; that is, it should not spuriously render such tools and approaches inapplicable.
- *Code compactness:* For each new version, it should be possible to (automatically) deploy code that pertains only to execution of that version, avoiding needless expansion of the deployed code.
- *Supermodel compactness:* The approach should not require or encourage replication (e.g., "copy and edit") of source lines, yet it should still allow maintenance (bug fixes and evolutionary changes) of both the normal system behaviors and the upgrade behaviors as well. In particular, it should avoid the quadratic blowup explained earlier, if possible. This is important, because run-time costs of validation tools can be proportional to supermodel size. For example, code coverage tools require enough test cases to cover all source statements.

Note that while in traditional software development, one usually validates the deployed code, I distinguish here between the representation of all system versions together for the purposes of predeployment validation (the *supermodel*) and the deployed code of a single version of the system (the *code*). While it may be that the supermodel is composed from the actual deployed code of the versions, this is not required; the supermodel could be expressed in a specification language which, after validation, is input to a code generator to produce the code. This is, in fact, the approach taken in the case study to be discussed later.

3 The Upgrade Framework

The key idea behind the *upgrade framework* is to explicitly represent source and data version numbers, upgrade transitions, and system versions together in one *supermodel* that enables behaviors that reach all legacy states. The supermodel is used during (predeployment) validation, and a tool is used to generate the (much smaller) code for a single version that is to be deployed to users. The framework consists of structural elements and structural conventions that govern the construction of the supermodel, as well as usage conventions that restrict the allowed sequences of input events to the supermodel. Taken together, these elements and conventions allow access to the different version behaviors and transitions between them, while encouraging representational efficiency and enabling a variety of validation techniques.

Before defining the framework, it is useful to develop some terminology for reactive systems.

3.1 Reactive Systems Terminology

Each reactive system is a state machine, whose state space may be finite or infinite. For concreteness, I will introduce a notation for describing reactive systems; however, it should be understood that this notation can be easily translated into whatever formalism is native to the software development environment used for a given project. I will represent reactive system state in terms of a collection of *indexed state variables (ISVs).* An indexed state variable $x : T_1, ..., T_{k-1} \mapsto T_k$ associates in a state s an index

$(k-1)$-tuple $[y_1, ..., y_{k-1}]$ to a value y_k, where each y_i is of type T_i. For example, in the email domain, we might define the ISV

$$\texttt{address} : \texttt{String} \mapsto \texttt{String}$$

which tells us the email address given a correspondent's nickname. In a particular state, this ISV might have the *instances*:

```
address("Sam") := "sam@somewhere.com"
address("Bob") :=
    "bob@channels.research.att.com"
```

For ISVs having 0 index parameters, representing traditional state variables, I will often omit the (empty) index parentheses.

A reactive system changes state in response to an *input event*. An *input event* is simply a labeled k-tuple of values, where the ith entry has type T_i and the label names the event type. In the email domain, we might define

$$\texttt{RECEIVE} : \texttt{Message}, \texttt{String}$$

which could have instances

```
RECEIVE(<email_message_1>, "Bob")
RECEIVE(<email_message_2>, "Sam")
```

that represent the reception of messages by users.

I will assume that the behavior of a reactive system in a given state in response to a given input event is defined in a *handler* for that event. This handler may take different syntactic forms in different reactive system representations; for example, in programming language code this might be a function, procedure, or method body. In a finite state formalism, on the other hand, this might be represented as a table entry. In any case, I will assume only that the handler has some way to receive the event parameters, compute and set state variables, and possibly yield outputs. For example, the following handler excerpt sets the inbox of the given user to the result of appending the received message to the previous inbox.

```
(defhandler (RECEIVE (Message ?msg)
                     (String ?user))
  SET(Inbox(?user),
      append(LOOKUP(Inbox(?user)), ?msg))
  ...)
```

Finally, I will assume (without loss of generality) that a reactive system has a unique *default initial state*, which is unconditionally reached in response to the distinguished `Init()` input event. Thus, the *reachable* portion of a reactive system's state space consists of precisely all states that can be reached by some sequence of input events beginning with `Init()`.

3.2 Upgrade Framework: Structural Elements

As alluded to above, the basic elements of the framework are relatively simple. The supermodel is itself a reactive system built from the reactive systems corresponding to the versions. This subsection describes the distinguished structural elements of the supermodel. I assume without loss of generality that the variables and events defined here do not conflict with any ISVs or input events of the target reactive system versions. (This can be enforced by renaming them if necessary.)

First, within the supermodel definition, we declare two ISVs and two input events:

- `SV`: $()\ \mapsto$`Integer` represents the current *source version* number. Intuitively, this number represents which version governs the behavior of the supermodel in the current state. The value of `SV` may change only in response to `SourceVersion` events:
- The `SourceVersion(x:integer)` input event indicates to the supermodel that its source version (`SV`) should be changed to `x`. The event handler for `SourceVersion` must do nothing other than `SET SV := x`.
- `DV`: $()\ \mapsto$`Integer` represents the current *data version* number. Intuitively, this represents the version of the source definition to which the current data state of the reactive system conforms. (I will expand on what this means below.) This is initialized to `SV` by each `Init` event and otherwise changes only in response to `Upgrade` events:
- The `Upgrade()` input event indicates that the system should upgrade its data version to (some) higher version number, performing any necessary insertions, transformations, or deletions of ISVs and their values. Among the state changes that must occur is the setting of `DV` to a value appropriate to the newly established state. Rather than specify that `Upgrade` must result in `DV` $=$ `SV`, which might seem reasonable at first glance, I require only that *if* `DV` $<$ `SV` prior to the `Upgrade` event, *then* the value of `DV` after the `Upgrade` event must be larger than it was prior to the event, but no larger than `SV`. Further, if `DV` $=$ `SV` *prior* to the `Upgrade` event, then no change must occur to the state at all.

Intuitively, each source version of the system will expect and maintain a certain set of semantic constraints and intended denotations of the data state. These may be explicit, implicit, or merely exist in the mind of the implementor, and the system version may or may not be successful in maintaining them. However, each state in the supermodel's state space is tagged with a DV value indicating which version's state space it belongs to. Normal (intra-version) events must not change DV; this prohibits the supermodel from switching between subspaces except at Upgrade or Init events.

3.3 Upgrade Framework: Structural Conventions

Structural conventions help us guarantee that we do not introduce spurious behaviors during the construction of the supermodel.

The first such convention, SC0, is simply that all normal (non-Upgrade and non-Init) event handlers must have the form:

```
(defhandler (NormalEvent <parameters>)
  (case (lookup SV)
    (range-1 <behavior-1>)
    ...
    (range-i <behavior-i>)
    ....))
```

where each range-i is of the form $(l_i..r_i)$, with $0 \leq l_i \leq r_i$. Each such range represents a range of nonnegative integers. The ranges within a case statement must be pairwise disjoint and their union must be all defined version numbers, i.e., from 0 up to the maximum version number included in the supermodel. Each behavior-i is a normal handler body, which could be programming language code, an executable model definition, or a specification. The case statement allows us to share behaviors between versions if nothing is changed in later version(s) of the handler. By requiring the ranges be declared using manifest constants, we allow simple automatic syntactic checking to guarantee the supermodel has a well defined behavior for any legal value of SV.

The second convention, SC1, is simply that *the* SourceVersion *event handler must set the* SV *variable equal to its input parameter and do nothing else*. Since SourceVersion events are undefined in the individual versions, there would be no advantage to allowing other behavior.

The third structural convention, SC2, decrees that *the* Init *event handler must have the following form:*

```
(defhandler (Init)
  (set DV (lookup SV))
  (case (lookup SV)
    (range-1 <behavior-1>)
    ...
    (range-i <behavior-i>)
    ....))
```

where the ranges are as described above. Note that the first thing done by the Init handler must be to set DV = SV.

```
(defhandler (Upgrade)
  (case (lookup DV)
    (range-1
      (case (lookup SV)
        (range-1-1
              <behavior-1-1>
              SET DV := <new-value-1-1>)
        ...
        (range-1-k
              <behavior-1-k>
              SET DV := <new-value-1-k>)))
    ...
    (range-i
      (case (lookup SV)
        ...
        (range-i-j
              <behavior-i-j>
              SET DV := <new-value-i-j>)
        ...))
    ...))
```

Figure 2. Required form of Upgrade handler.

The fourth and final structural convention, SC3, describes the form of the Upgrade event handler. Essentially, it is a nested case statement structure, reflecting the fact that different upgrade behaviors may be necessary for different combinations of SV and DV. See Figure 2.

Note that each subcase must end by setting DV to a manifest constant (new-value-i-j). This allows a simple automatic syntactic check to verify that Upgrade events result in an allowable increase in DV. This, combined with checking the ranges for appropriate coverage and nonintersection, enables us to ensure that the supermodel's response to an upgrade event does increase DV but not more than SV. That is, we enforce that new-value-i-j is greater than every element of range-i and less than or equal to all elements of range-i-j.

Figure 3 shows a schematic representation of a typical Upgrade handler. To compute the supermodel's response to an Upgrade event, we simply look up in the table the square corresponding to the current (SV, DV) pair. If this is a handler body, it is executed and sets DV accordingly; if it is an arrow, then the square pointed to by the arrow is examined, etc. A typical case is when the only non-arrow entries are at positions (i+1, i), however our more general form allows for later source versions to "patch" the upgrade behaviors of earlier versions by having new definitions for the upgrade handlers. For example, in the figure, an Upgrade event when SV = 4 and DV= 0 results (after following the arrows) in executing the handler defined in the (1,0) box, which in turn ends up setting DV to 1. The entry in box (4,1) is a version-4 patch to the upgrade behavior for cases when DV = 1, resulting in a direct transition to DV =

3, instead of the transition to `DV` = 2 which was in effect for source versions less than 4. Figure 4 shows the corresponding `Upgrade`-induced transitions among the version subspaces implied by the table in Figure 3.

This representation of the `Upgrade` handler has two benefits. First it allows *patching* the `Upgrade` transitions by adding transition handlers in later source versions that override those in earlier source versions. As will be discussed later, we can still generate tests that reach legacy instances built by earlier source versions, yet allow later source versions to `Upgrade` using a patch that is more desirable (e.g., less buggy).

And yet, we retain this flexibility without incurring the quadratic costs (in the vast majority of practical cases) of storing all source versions. The second benefit of this representation, therefore, is its compactness. The outer case statement dispatches on `DV`, because it is quite unlikely that we would wish to perform exactly the same `Upgrade` operations on two different data versions. By contrast, it *is* quite likely that adjacent source versions will perform identical `Upgrade` actions, so we specify that the inner case statements dispatch on `SV` to allow maximum sharing. In the most common case, a supermodel will contain no "patches" to the `Upgrade` handler, so the `Upgrade` handler will have just one handler fragment for each transition from i to $i+1$. This is only linear in the number of versions, which is superior to the quadratic requirement of the naive store-all-versions approach. When patches are necessary, they only require space for a single copy of the patch transition, instead of a copy for each succeeding source version.

Note that the worst case (most squares filled) is still quadratic, but this now rarely if ever arises in practice, whereas the store-all-versions approach is quadratic in *every* case.

3.4 Upgrade Framework: Usage Conventions

The first usage convention, UC0, states that (a) every `SourceVersion` event is immediately followed either by an `Init()` event or else by a sufficient number of `Upgrade()` events to guarantee that `DV` = `SV`; and (b) the first event applied each time the software is activated is a `SourceVersion` event. In short, we adopt a policy of "upgrade completely first", both for deployed code and during validation of the supermodel. In pseudocode:

```
SourceVersion(x);
Init();

     OR

SourceVersion(x);
while(DV < SV) Upgrade();
```

The primary advantage of UC0 is that it allows us to use a tool that generates production code from the supermodel

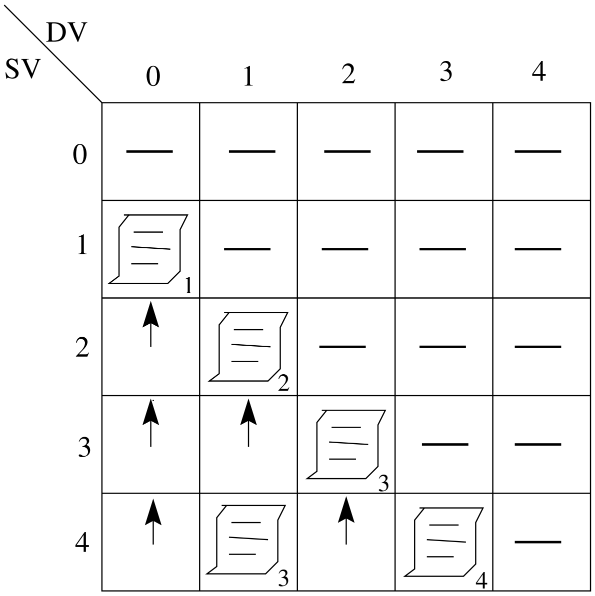


Figure 3. A schematic diagram of a typical `Upgrade` event handler. A number in the lower right corner of a box is the new value of `DV` after handler execution. Boxes with dashes represent cases disallowed by usage conventions.

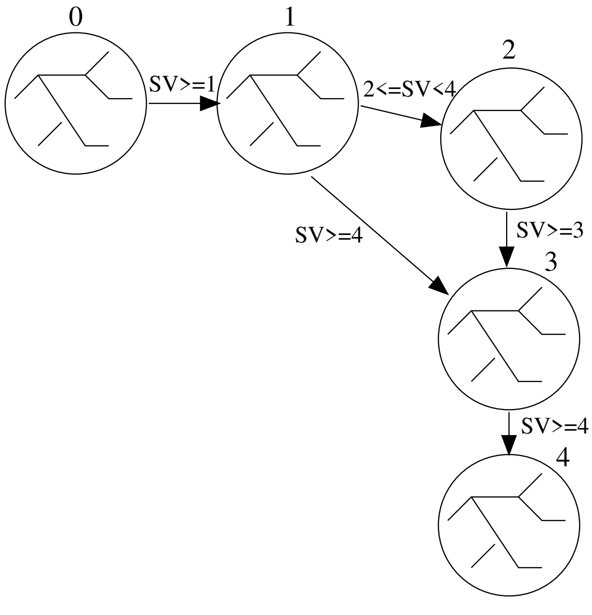


Figure 4. The `Upgrade` transitions among version subspaces corresponding to the diagram in Figure 3.

that contains only those portions of the event handlers relating to the newest version. In compiler terms, when deploying version n, it allows the code generator to assume that `SV` $= n$ and do constant folding and dead code elimination[10]. This implies that the deployed code size is no bigger than it would be in just deploying version n in the traditional way.

The advantage of obeying UC0 during supermodel validation is simply that this is the way the deployed versions behave, so there is no need to validate behaviors that violate UC0. We can, of course, validate behaviors that apply normal events when *both* `SV` and `DV` are less than the current (maximum) version number accommodated by the supermodel; we only disallow normal events when `SV` $>$ `DV`.

The second usage convention, UC1, simply states that *the integer argument to* `SourceVersion` *events may not be less than the current value of* `SV`. In short, "downgrading" is disallowed. While it is possible to extend the framework to allow this, we would not be able to enforce that `Upgrade` events always increase the value of `DV` (because they wouldn't). This could lead to programming errors in upgrades that would otherwise be caught, not the least of which are infinite loops caused by UC0. On the other hand, there is little benefit to handling downgrading, given how infrequently one wishes to downgrade one's system.

The third usage convention, UC2, is a methodological one. We require that once a version is deployed, all `case` branches covering that version number may no longer be altered. In other words, the next version of the system must be constructed by either adding an entirely new case or by adding the new version number to an existing case selection range. This convention guarantees that we will always be able to generate a test to reach any legacy state that was reached by a deployed instance.

4 Supermodel Validation Techniques

In this section, I will illustrate the validation techniques that can be applied to a supermodel conforming to the upgrade framework.

4.1 Testing

Reachability. Recall that one problem with the traditional test-what-you-deploy approach is that legacy states are unreachable and so no test can be generated to cover these cases. If, however, all deployed system versions are generated from a supermodel conforming to UC2, then we can generate tests for the supermodel that reach those states.

To illustrate this, suppose we deploy source version 2 of a system to a user who initializes it, applies three normal input events (`Event1("Bob")`, `Event2("Bob", "Alice")`, `Event3()`), upgrades to version 4, and then applies two more normal input events (`Event4("973-360-8207")`, `Event5()`). How can we test whether version 7 will correctly upgrade that user's legacy instance? We simply use the `SourceVersion` and `Upgrade` events of the supermodel to model the actions of installing new versions and can reach any state reachable by the user. Here is a supermodel test case that reaches it:

```
SourceVersion(2);
Init();
Event1("Bob");
Event2("Bob", "Alice");
Event3();
SourceVersion(4);
Upgrade();
Upgrade();
Event4("973-360-8207");
Event5();
```

The tester may prepend this event sequence to any desired tests of the `Upgrade` behaviors for source version 7. For example, one might extend it with `SourceVersion(7); Upgrade(); Upgrade(); Upgrade(); Event6(); <observations>`

It is likely that testing upgrade behaviors involves more test cases than testing only normal behaviors of a single version. A contribution of the upgrade framework is that we can now approach testing upgrade behaviors in the same way as testing other system behaviors using the supermodel.

Assertions. Of course, it is clear that since we can now generate tests to reach states accessible from legacy instances, we can use assertion checking[8] to monitor correctness conditions, invariants, etc. These assertions may be inserted directly in the source of the versions, or may be added externally by the validator, using tools. For example, in our email domain, we might be interested in checking that after upgrading to a centralized list of all nicknames in source version 7, every nickname in the `nicknames` list has a nonempty corresponding email address; that is

$$\forall n \in \texttt{(lookup nicknames)}.\texttt{(lookup addr(}n\texttt{))} \neq \texttt{""}$$

Using well known techniques, we could add such an assertion to the source of the `Upgrade` handler for transitions to `DV`$= 7$.

Coverage. Since the supermodel is simply another reactive system of the same kind as those of the versions, we can apply the same coverage metrics and tools to it as we can to the versions. In particular, since now the `Upgrade` handler code branches are reachable, we can hope to achieve 100% coverage.

4.2 Model Checking

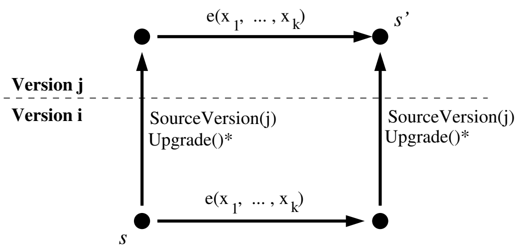
Since now states reachable only from legacy instances are formally reachable, model checkers[5, 1] can be effective in finding errors in `Upgrade` behaviors. Of course, model checkers are subject to the state explosion problem, and by connecting the state spaces of the versions together we create a much larger space. However, this is inherent to any approach to model checking `Upgrade` behaviors, since one must have a model in which legacy states are reachable and in which states reachable only by upgrading are also reachable. One may limit the model to a two-version submodel of the supermodel, but this risks missing some otherwise reachable states.

4.3 Theorem Proving

Formal proof (automated or otherwise) is not really helped by the upgrade framework, except that it provides an integrated single model on which to apply tools. Anything one can prove of the supermodel could in principle be proven by breaking it into proofs over each of the versions.

However, there are classes of theorems which, if proven of the supermodel, can potentially drastically reduce the validation effort. For example,

> **Theorem 1:** Let n be the maximum version defined in the supermodel. *If* (1) for all versions $i < j \leq n$, for all non-`SourceVersion`, non-`Upgrade`, and non-`Init` input events $e(x_0, ..., x_k)$ of version i, and for all states s in the reachable state space of version i, the following state diagram commutes (n.b.: `Upgrade()*` denotes `while(DV < SV) Upgrade();`)



> *and* (2) for all $i < j \leq n$ `SourceVersion(i); Init(); SourceVersion(j); Upgrade()*;` results in the same state as `SourceVersion(j); Init();` *then* all states reachable via upgrading legacy states are also reachable within the state space of version n.

The first condition just says that all normal events commute with `Upgrade` events, while the second says that upgrading the default initial state results in the default initial state of the higher version. Together, these imply that the validation problem can be drastically reduced to considering only the reachable space of the latest version. Indeed, the space-multiplication problem (see Section 1.1) does not arise, because lower versions' spaces are embedded within the reachable portions of higher versions' spaces.

Proof sketch. We prove it here for the case of one upgrade, but induction can extend this to the general case. Suppose the legacy instance is reached in the supermodel by the input event sequence `SourceVersion(i); Init();` $e_{i1}, e_{i2}, ..., e_{ik}$`;` `SourceVersion(n);` `Upgrade()*;` $e_{n1}, e_{n2}, ..., e_{nl}$`;`, where the e_{ij} are normal events. By commutativity, the same state is reached by the event sequence: `SourceVersion(i); Init(); SourceVersion(n); Upgrade()*;` $e_{i1}, e_{i2}, ..., e_{ik}$`;` $e_{n1}, e_{n2}, ..., e_{nl}$`;`, which can be replaced (via the second hypothesis) with `SourceVersion(n); Init();` $e_{i1}, e_{i2}, ..., e_{ik}$`;` $e_{n1}, e_{n2}, ..., e_{nl}$`;`. The latter is a scenario in version n's reachable space. □

Here is an upgrade transition handler from the email agent case study which upgrades from version 12 to 13:

```
((12)
 ;added encrypted correspondent-notes field
 (LOOP
   (FOR (STRING ?CORR) IN
        (LOOKUP KNOWN-CORRESPONDENTS))
   (DO
       (SET ENCR-CORRESPONDENT-NOTES ?CORR
            "")))
 (SET CONTROLLER-VERSION 13))
 ; "SET DV := 13" in framework terms
```

This is simply adding a new data member (the "encrypted notes" field) for each known correspondent in the database. The field is initialized to the empty string. The only way to set this field to a different value is via the `SET-CORRESPONDENT-NOTES` input event, which is new for version 13. Thus, all version-12 events must commute with this `Upgrade` action: if a version-12 event occurs first, then the `Upgrade` will initialize the new field as above; otherwise, the `Upgrade` event will occur first and then the version-13 handler for the version-12 event will simply handle the event in the same way as the version-12 handler, i.e. without altering the new field at all, since only the new version-13 event can alter it. Of course, to apply the theorem, we must prove the same thing of all earlier upgrade transitions as well as the condition on upgrading the default initial state. But this gives a flavor for the theorem's use in practice. (Note that the inner `case`, discriminating on `SV`, has been suppressed since it would have only one branch.)

The reachability property above is not conclusive of correctness of the `Upgrade` transitions, since the result could be reachable but incorrect. However, it follows from the commutativity property that all we need to do is to re-run a regression validation suite and verify it is unchanged in the new version: upgrading any correct old-version behavior leads to the same results as if we had carried out the entire scenario in the new version. Thus, the upgrade step must have been correct.

Commutativity is a fairly stringent requirement that may not hold for a reasonable design. For example, in the file security example, when upgrading to a new encryption cipher, I may choose to leave pre-encrypted files alone, marked as using the older cipher, while all subsequently encrypted files would be encrypted with the new cipher. Thus, the `EncryptFile` event would not commute with the `Upgrade` event. For such systems, there are generalizations (weakenings) of Theorem 1 that may be practically as useful. For example, we can get a similar result if we only prove that there is some sequence of new-version events that reaches the same state as the upgrade of a state reached from an old-version event. (In Theorem 1 this sequence must consist of exactly the same single event.) Space is too tight to explore that theorem here, however.

5 Update Framework Evaluation

I believe the upgrade framework meets the requirements stated in Section 2:

- *Wide applicability.* Since the structural elements and conventions are expressed in terms of the same constructs as are used to implement the reactive systems themselves (e.g. program code, specification language) and since simple tools can generate essentially the same code as would be deployed using traditional methods, this approach is applicable to most imaginable reactive system development environments.

  For example, in a C language development environment, the `case` statements become either C `switch` statements or C-preprocessor `#if` statements, input events are applied as function calls, and ISVs are implemented as either array or scalar variables; code version generation could be handled using C-preprocessor directives. Supermodel syntax and range checking tools are simple to implement using freely available C syntax parsers.

- *Testability.* Section 4 showed how the supermodel-based framework allows testing, assertion checking, and coverage tools to be applied to upgrade behaviors.

- *Formal methods applicability.* Again, since the supermodel is constructed in a constrained way out of the same elements as are present in the target system formalism, formal methods previously applicable remain applicable. In addition, model checkers can now detect errors in upgrade behaviors, since they are now in a reachable portion of the state space. This point is subject to the inherent complexity limitations of model checking induced by the size of the combined state space of two adjacent versions.

  Finally, while theorem provers are no less applicable to the supermodel, and possibly a bit easier to apply, I showed how they could be used in conjunction with Theorem 1 (and its variants) to greatly simplify the validation problem. The proofs supporting the use of Theorem 1 need not be automated, either; it can be heuristically valuable to do an informal proof of the commutativity property when proof automation is not practical or available.

- *Code compactness.* As discussed above, simple tools can be used to generate the same size code as would be produced for a single version traditionally.

- *Supermodel compactness.* The representation of handlers, particularly the `Upgrade` handler, greatly reduces the size of the supermodel, particularly compared to the quadratic typical case of the store-all-versions approach. Even so, it still allows flexible patching of behaviors by later source versions.

To gain further experience with and evaluate the upgrade framework, I have applied it to a moderate-sized reactive system, the Email Channels system controller. The Email Channels system[3] is a novel spam-free email system. It is being used currently in a small trial with computer-naive users which started in July, 1999. A fuller description of the formal methods techniques applied in its development is forthcoming[4]. It has evolved through 16 versions during the trial, though only versions 9 through 16 are modeled using the upgrade framework. Since the users depend on their Channels components for their main email service, `Upgrade`-induced errors would be unacceptable. Over the course of the trial, users upgraded their legacy instances (each instance containing a state database of approximately 5 kB of information) at different rates and times. Currently, many users are at version 15 and one is still at version 8.

In using the supermodel approach, I discovered a subtle error, having to do with relative ordering of `Upgrade` and user initialization events, that could have led to component failure in some cases. This error was found and corrected at the specification level (i.e. not after deployment, where it would be much less acceptable and more expensive to fix). I found it using coverage analysis of the `Upgrade` behaviors, which are unreachable (hence untestable) in a single version of the system. Thus, the supermodel approach was crucial in finding this problem.

Regarding supermodel size, the `Upgrade` handler is 178 lines long. If I had used the store-all-versions approach, it would have been 1155 lines long. This is a huge savings in code coverage testing costs alone, since I used 100% branch coverage as one validation criterion during the case study.

The `Upgrade` handler has 11 branches, while that of the store-all-versions approach would have had 71.

6 Related Work

While there has been much work done on the general issue of system evolution, little is relevant to the state upgrade problem. Cook and Dage[2] handle component upgrading by maintaining running old and new versions of the component, with a declarative technique for defining in a given situation which of the versions is "authoritative". If the versions differ, then the authoritative one is believed. This addresses the problem of upgrades causing unintended changes to behaviors intended to remain unchanged; however, it does not address the problem of mapping the legacy state to an appropriate new state when a change *is* intended.

Little and Shrivastava[7] describe a technique for reconfiguring distributed systems by managing object replication; however, reconfiguration is achieved by duplicating objects. This does not address evolving object states.

Regression testing[6] is about executing tests to verify that desirable functionality is unchanged across a version change. The state upgrade problem, by contrast, involves state changes leading to new and different functionality.

Software version control systems such as RCS[9] provide storage of multiple versions and automatic tools for generating a single deployment version. They exploit essentially the same idea behind the compactness of the `Upgrade` handler: storing only the changes between versions instead of the entire version. The upgrade framework as described only allows changes at the level of handler bodies, rather than at the much finer grained level of source lines. On the other hand, version control systems do not explicitly address the problem of executing the implicit supermodel at all, much less for the purpose of dealing with the state upgrade problem. The present work could be viewed as an extension of version control systems with validation techniques for the state upgrade problem.

7 Limitations and Future Work

The upgrade framework as described cannot handle changes to the signature of a given event type. Instead, I define a new event with the new signature. However, the framework could be extended to handle changing event signatures by conceptually placing the `case` statements dispatching on `SV` *outside* the handler definition entirely. Actual development environments would likely implement this using either polymorphism or source macros.

The framework as described doesn't address how to manage changes to support functions that are used by the handler bodies (e.g. library functions). I currently adopt naming conventions and essentially give new versions of the library functions related names, keeping the old definition in the library for use by the supermodel in executing older versions. This has not been a significant difficulty in the case study, but may need to be improved in the future.

8 Conclusion

The upgrade framework, based on the supermodel representation for reactive system versions, is simple and yet highly effective in allowing both compact representation and thorough validation of state upgrade behaviors. It can be viewed as an extension of software version control systems to handle validation. Its simplicity allows it to be implemented within virtually any software development environment with relatively little effort. Thus, I believe the idea deserves consideration for mainstream use by developers of reactive system products.

References

[1] J. Burch, E. Clarke, K. McMillan, D. Dill, & L. Hwang; Symbolic model checking: 10^{20} states and beyond; *Info. and Comput. 98*, 142–170, 1992.

[2] J. Cook & J. Dage; Highly reliable upgrading of components; In *Proc. 21st Intl. Conf. on Software Eng.*; New York, NY: ACM, 203–212, 1999.

[3] R.J. Hall; How to avoid unwanted email; *Comm. ACM 41(3)*, 88–95, March 1998.

[4] R.J. Hall; Specification, validation, and synthesis of email agent controllers: a case study in function rich reactive system design; in *Proc. 3rd Workshop on Formal Methods in Software Practice*, 2000, ACM.

[5] G.J. Holzmann; *Design and validation of computer protocols*; Englewood Cliffs, NJ: Prentice Hall, 1991.

[6] B. Korel & A. Al-Yami; Automated regression test generation; In *Proc. ISSTA-98 (Software Engineering Notes 23(2))*; New York, NY: ACM; 143–152, 3/1998.

[7] M. Little & S. Shrivastava; Using application specific knowledge for configuring object replicas; In *Proc. 3rd Intl. Conf. on Configurable Distributed Systems*; IEEE Computer Society Press, 169–176, 1996.

[8] D. Rosenblum; A practical approach to programming with assertions; *IEEE Trans. on Software Eng*, 1995.

[9] W. Tichy; RCS – a system for version control; *Software Practice and Experience 15(7)*; 637–654; 1985.

[10] J. Ullman, A. Aho, & R. Sethi; *Compilers: Principles, Techniques, Tools*; Addison-Wesley Longman, 1986. ISBN: 0201100886